UC Irvine
ICS Technical Reports

Title
A performance comparison of several superscalar processsor [sic] models with a VLIW processor

Permalink
https://escholarship.org/uc/item/1kg8b61b

Authors
Lenell, John
Bagherzadeh, Nader

Publication Date
1992

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ARCHIVES
Z
699
C3
no.92-92

A Performance Comparison of Several Superscalar ProcesssorModels with a VLIW Processor

John Lenell and Nader Bagherzadeh

Department of Electrical and Computer Engineering
Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No. 92-92

A Performance Comparison of Several Superscalar Processor Models with a VLIW Processor

John Lenell and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717

Abstract

Superscalar and VLIW processors can both execute multiple instructions each cycle. Each employs a different instruction scheduling method to achieve multiple instruction execution. Superscalar processors schedule instructions dynamically, and VLIW processors execute statically scheduled instructions. This paper quantitatively compares various superscalar processor architectures with a Very Long Instruction Word architecture developed at the University of California, Irvine. An architectural overview and performance analysis of the superscalar processor models and VIPER, a VLIW processor designed to take advantage of the parallelizing capabilities of Percolation Scheduling, are presented. The motivation for this comparison is to study the capability of a dynamically scheduled processor to obtain the same performance achieved by a statically scheduled processor, and examine the hardware resources required by each.

1 Introduction

RISC microprocessors achieve high performance by executing close to one operation per cycle, and employing aggressive technology dependent hardware techniques. These techniques decrease the processor cycle time, thereby reducing the time to perform a task. As the rate of performance improvement due to technological advances subsides, other methods for improving processor

performance must be developed. Exploiting parallelism within the instruction stream is one method for improving processor performance. Utilizing instruction parallelism is achieved by executing multiple instructions concurrently. A microprocessor executing more than one instruction each cycle improves performance by reducing the number of cycles required to execute a program.

VLIW and Superscalar processors are two promising design techniques for executing more than one instruction each cycle. Both architectures are RISC-like with multiple pipelined execution units for executing instructions in parallel. However, each architecture exploits the instruction parallelism in a different manner.

The VLIW architecture requires statically scheduled code for instruction parallelization. Trace or Percolation Scheduling(PS) compilers perform global code optimization for the VLIW and schedule the optimized code into groups of operations which are fetched simultaneously as one instruction [5, 9]. Each operation in the instruction word controls a single execution unit. All of the operations in a VLIW word are executed in parallel, and results are written to a globally shared register file[3].

Alternatively, a superscalar processor provides complex hardware resources to detect and issue parallel instructions dynamically as they are fetched from a linear sequence of instructions. All instructions issued in a cycle are executed in parallel, and a global register file is updated with the instruction results. Some superscalar architectures maintain the order of the instruction stream at issue, while more elaborate hardware can be added to allow out-of-order instruction issue. Additionally, the superscalar processors execute speculatively to increase the number of instructions available to issue, and reduce the delays associated with conditional branches. Performance of the superscalar processor generally increases with the complexity of the hardware as it attempts to look farther ahead into the instruction stream to issue and execute independent instructions, out of order, and speculatively.

The performance of both processor architectures is limited by data hazards, control hazards, and available resources. An instruction scheduler attempts to free an instruction from hazards and resource conflicts so it can be issued to an execution unit in parallel with other instructions, yet independent of the operation and result of the other instructions. The performance of a processor is dependent upon the performance of the instruction scheduler. Instructions can be scheduled statically (during compile time) or dynamically (at run time).

1.1 Instruction Scheduling

Static compilation requires a complex compiler to exploit a large amount of instruction parallelism by scheduling beyond basic blocks. The compiler attempts to compact the output code for the target VLIW architecture so that each instruction word field is occupied by an operation from the original program sequence. In so doing, the compiler is constrained by data dependencies, control dependencies, resource conflicts, and register usage. As a result, the compiler is forced to schedule no-ops in fields it has been unable to schedule valid operations. The use of no-ops in VLIW's reduces the average number of operations executed per cycle, and can greatly increase the size of the compiled code.

Dynamic scheduling in superscalar processors refers to the ability of the hardware to detect and issue multiple instructions at run time. An effective dynamic scheduler is important for a superscalar processor to maintain parallel instruction execution. Numerous dynamic scheduling techniques have been explored [13, 12, 14, 11]. The advantages of dynamic scheduling are the following [6]:

- Efficient scheduling of dependencies unknown at compile time.
- Simplifies compiler design.
- Maintains code compatibility between generations of processors.
- No code expansion due to scheduling.

Unfortunately, these advantages are gained at significant hardware expense. This expense is mitigated by limiting the number of instructions which can be scheduled for execution each cycle. As a result, dynamic scheduling is disadvantaged by its inability to perform global code scheduling as is done by static scheduling.

The dynamic scheduler may be designed to issue instructions in their correct program sequence, or lookahead into the instruction stream and issue instructions out of their original order. These issue policies are known as *in-order issue* and *out-of-order issue* respectively[7]. An in-order-issue policy is the simplest to design, but its performance suffers in the presence of hazards and resource conflicts. Instructions are issued in program order from the decoder until an instruction has a hazard or resource conflict with a preceding instruction.

Instruction execution bandwidth can be increased by allowing the processor to continue to issue instructions which follow stalled instructions. This implies an out-of-order issue policy because the original program sequence will not be maintained. An instruction window is used to hold instructions after they have been decoded and are waiting to execute. All of the instructions in the instruction window are available to the issue unit. This allows the issue unit to lookahead in the instruction sequence and issue the maximum number of instructions to the execution units.

The purpose of this research is to compare these two processor design alternatives by analyzing performance and hardware requirements. For this purpose, a scalable instruction-level processor simulator has been developed to evaluate the performance of superscalar models. The simulator has been designed to explore the performance of both in-order issue and out-of-order issue policies, as well as, the influence of the size of critical hardware elements on the performance of the processor. These superscalar models are compared with the VIPER, a VLIW processor, which has been developed at the University of California, Irvine[1]. An overview of the VIPER architecture is given in Section 2. The superscalar model is presented in Section 3, and the following section explains the simulation methods. Section 5 discusses the simulation results of the VIPER and superscalar models.

2 The VIPER Processor

VIPER is an integer VLIW processor which fetches a single long instruction specifying four operations each cycle. The operations are independent and execute in parallel on four *functional units*. A functional unit consists of one or more execution units. Each functional unit includes an arithmetic/logic and either a load/store or control transfer execution unit. The arithmetic/logic execution unit (ALU) is capable of executing all simple integer operations including shift and compares. The load/store execution unit (LS) provides the off-chip memory interface. The control transfer unit (CT) provides the function of altering the program counter address as a result of control transfer operations. Two ALU/LS functional units and two ALU/CT functional units are used on the processor. Each functional unit can read two 32-bit operands from a multi-ported global register file containing 32 registers. The operation set and pipeline structure are discussed below. A more detailed description of the VIPER architecture can be found in Reference [1].

2.1 Operation Set

VIPER implements a RISC-type operation set. An operation is specified by a 32-bit field of the instruction word. A total of 29 operations are defined for the processor. These operations are divided into arithmetic/logic, load/store and control transfer categories to facilitate the assignment of an operation to a functional unit. The complete operation set is shown in Table 1.

The arithmetic/logic category defines the arithmetic, logic, comparison, and shift operations. The operations are executed by the ALU execution units.

Two instructions, load word (LDW) and store word (STW), are defined in the load/store category. These operations perform register indirect loads and stores, and are executed by the LS units.

Both conditional and unconditional branches are defined in the control transfer operation category. Two types of unconditional branches are specified, CALL and JUMP. These instructions cause the program counter to be loaded with a target address. The target can be specified with a register or immediate value. CALL instructions are used for procedure calling. They write the return address into register 31. JUMP operations are a more general unconditional branch and only cause the program counter to change to the specified address.

Conditional branches are performed with advanced conditioning. Conditions are set with explicit compare instructions. The compare instructions set the least significant bit of a specified register which becomes the branch condition code. VIPER can perform multi-way branches by testing two condition codes per branch operation, and executing branches on up to two functional units simultaneously[2]. The branch operations have the following form:

$BRc_1c_2 \;\; cc_1, cc_2, offset$

where c_1 and c_2 are conditions having a true or false value. The condition codes are the least significant bit of cc_1 and cc_2 which each specify one general purpose register. The offset is added to the value of the program counter to form the target address. Four conditional branch operations are available for testing the four possible conditions. Three-way branches can be effected by executing two branch operations together. Three targets can be generated in this case. If the first branch test succeeds then the target becomes the program counter plus the offset of the first branch. Otherwise, if the second branch succeeds, the program counter is added to the offset of the second branch. If

Operation	Operands	Description
ADD	*src1,src2,dest*	integer add
SUB	*src1,src2,dest*	integer subtract
AND	*src1,src2,dest*	logical AND
OR	*src1,src2,dest*	logical OR
XOR	*src1,src2,dest*	logical exclusive-OR
NOT	*src1,dest*	bitwise complement
LSL	*src1,src2,dest*	logical shift left
LSLI	*src1,#SA,dest*	logical shift left (constant)
LSR	*src1,src2,dest*	logical shift right (variable)
LSRI	*src1,#SA,dest*	logical shift right (constant)
ASR	*src1,src2,dest*	arithmetic shift right (variable)
ASRI	*src1,#SA,dest*	arithmetic shift right (constant)
SEQ	*src1,src2,dest*	set if equal
SNE	*src1,src2,dest*	set if not equal
SLT	*src1,src2,dest*	set if less than
SLTU	*src1,src2,dest*	set if less than unsigned
SGE	*src1,src2,dest*	set if greater than or equal
SGEU	*src1,src2,dest*	set if greater than or equal unsigned
ADDI	*src1,#I,dest*	integer add immediate
SUBI	*src1,#I,dest*	integer subtract with immediate
LUI	*#I,dest*	load upper immediate
LDW	*src1,dest*	Load data word from M[src1]
STW	*src1,src2*	Store data word to M[src1]
BRFF	*src1,src2,#O*	branch if c1=false, c2=false
BRFT	*src1,src2,#O*	branch if c1=false, c2=true
BRTF	*src1,src2,#O*	branch if c1=true, c2=false
BRTT	*src1,src2,#O*	branch if c1=true, c2=true
CALL	*src1* or *#LI*	unconditional procedure call
JUMP	*src1* or *#LI*	unconditional jump

Table 1: VIPER Arithmetic/Logic Operation Set

Pipe Stage	Description
IF	Instruction fetch.
ID	Instruction decode and operand fetch.
EX	Operation execute.
WB	Result write back.

Table 2: VIPER Pipeline Stages

both branch operations fail, then the program counter is incremented to the next address.

2.2 Pipeline Structure

VIPER has a four stage instruction pipeline. The stages are shown in Table 2. Each stage completes in one cycle. The instruction fetch stage reads one long instruction word each cycle and latches the instruction into the decoder at the end of the cycle. An instruction cache miss will stall the instruction fetch mechanism.

Operands for the operations are obtained during the instruction decode stage from either the register file or from a functional unit through a bypassing network. Operations are distributed to their corresponding functional unit at the end of the cycle.

The execute stage performs all operations in a single cycle, and result bypassing between all functional units can occur during this stage to eliminate stalls due to data hazards. Control hazards are handled with a delayed branch of one cycle.

Results of the functional units are written to the register file during the write back stage. The write occurs during the first phase of the cycle, so a read can be made to the register during the second phase in the instruction decode stage.

2.3 The PS Compiler

A component of VIPER is the Percolation Scheduling compiler[10]. The compiler attempts to increase the parallelism available to the processor by compacting across basic block boundaries, performing loop pipelining, and register

renaming. The compaction process attempts to move operations as high as possible in the program by extending the instructions horizontally. The program is scanned in a top-down manner and instructions are moved up the program graph if the original semantics of the program can be maintained. The compiler also performs loop pipelining with a method called Perfect Pipelining. Perfect Pipelining is an algorithm which pipelines general loops, including loops with conditional jumps inside the loop body. Finally, the compiler eliminates *false* dependencies due to reusing registers by employing *register renaming* during the compaction process.

3 The Superscalar Processor Model

The superscalar model performs 32-bit integer operations. Multiple instructions are fetched each cycle, and the processor is able to issue and complete up to four instructions per cycle, requiring an eight read, four write port register file. The processor model has 32 general purpose registers. Memory is accessed through explicit load/store operations. Additionally, the following architectural features are defined for the processor model:

- Instruction Set
- Instruction Fetch Mechanism
- Branch Prediction

3.1 The Instruction Set

With the exception of the control transfer instructions, the instruction set of the superscalar model is identical to that of the VIPER processor. However, the VIPER processor is capable of performing multi-way branches which is a feature not supported by the superscalar model. Instead, the four branch operations of VIPER are replaced by two two-way branch instructions. The two branch instructions defined for the superscalar model are the following:

BRT src1,#0

BRF src1,#O

BRF and *BRT* perform branch if false and branch if true operations, respectively on the least significant bit of the register designated by the src1 field. The value *O* is an offset added to the current program counter to compute the branch target address.

3.2 Instruction Fetch Mechanism

Instructions are fetched n at a time (n is 2 or 4, depending on the simulation). The program counter is always a multiple of n, and contains the address of the first instruction to be fetched, and it is used to access a cache of line size n. A line of instructions is fetched from the cache and latched into the instruction decoder at the end of the instruction fetch stage. If a control instruction transfers instruction flow to an instruction other than the first of a line, the whole line containing the target instruction is fetched. The misalignment is compensated during decode by masking out instructions preceding the target of the control transfer.

3.3 Branch Prediction

A dynamic branch predictor is utilized to reduce the number of branch delay cycles and maintain instruction fetch bandwidth. Branch prediction is implemented with a branch target buffer (BTB)[4, 8]. The program address, branch target, and predicted direction of all control transfer instructions are stored in the BTB.

3.4 Machine Configurations

To complete the description of the processor model for simulation, a machine configuration is specified by:

- Set of Functional Units
- Dynamic Scheduling Technique
- Instruction Cache Interface

3.4.1 Functional Units

Like the VIPER, each processor uses a combination of three types of execution units for executing instructions after issue. They are ALU, LS, and CT as described previously in Section 2. Each execution unit can begin and complete one instruction per cycle. Several different configurations of these execution units will be examined during simulation, so that the configuration with the best cost/performance ratio can be determined.

3.4.2 Dynamic Scheduling Technique

One of the following three scheduling techniques can be selected for the machine configuration:

I-D This notation specifies a scheduler performing in-order issue from the instruction decoder. The decoder size is limited to the number of instructions fetched each cycle. Instruction fetch is stalled until all of the instructions have been issued from the decoder.

I-W In-order issue from a central instruction window is specified by this notation. The instruction decoder dynamically performs register renaming, and moves the instructions into the window. When the instruction window is full, the decoder stalls, and the instruction fetch is stalled until all of the instructions have been moved out of the decoder. Instructions issue from the window in the original program order. The number of instructions issued each cycle is determined by the number of available functional units.

O-W This notation specifies an out-of-order issue policy from a central instruction window. It also does register renaming during the instruction decode, but when the instructions issue from the window, they can be issued in any order.

3.4.3 Instruction Cache Interface

The instruction cache can be explicitly modeled by defining its miss ratio and miss penalty. Cache misses randomly occur at the rate specified by the miss ratio. A miss causes the instruction fetch to stall for the number of cycles

specified by the miss penalty. A 100% hit ratio is achieved by setting the miss ratio to zero.

In the following sections, the performance of several different machine configurations will be presented. The notation used for identifying the configuration is to give the scheduling technique, and the size of the instruction window if applicable. All other parameters will be explicitly presented.

4 Simulation Methods

Two simulators were used for comparing the performance of the superscalar and VIPER processors. This section describes the two simulators, and the benchmarks used during simulation are given.

4.1 The VLIW Simulator

A VLIW instruction level simulator is included with the Percolation Scheduling compiler to evaluate architectural alternatives. The simulation path for the VIPER processor is shown in Figure 1. Two files are specified as inputs to the simulation process, the benchmark program and the hardware configuration file. The target architecture, VIPER, is defined in the hardware configuration file. Each benchmark is compiled by GCC into an intermediate code which is independent from any machine architecture. The intermediate code is represented by a Control/Data Flow Graph (CDFG) with each node containing one operation. The fourth step compacts the operations of the original CDFG into a CDFG with multiple operations in each node. The target architecture is specified for this process to provide resource constraint scheduling. Code generation uses the hardware configuration file and produces a machine program which can be executed by the target architecture. An assembly language program is the final output of the code generator. Finally, the hardware configuration and the assembly language program are used by the simulator to compute the execution time of the benchmark. The VLIW simulator provides the following outputs:

- The number of cycles taken to execute the program.
- The frequency of individual operations.
- The number of no-ops executed dynamically.

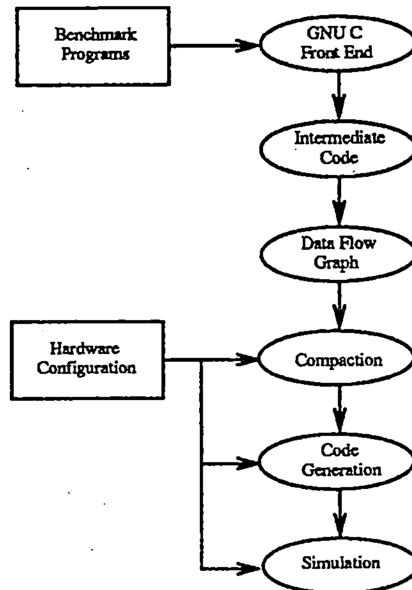

Figure 1: Simulation Path for VIPER

4.2 The Superscalar Simulator

A scalable and reconfigurable simulator has been developed to evaluate the performance of the superscalar models described in the previous section. The simulator executes the input program at the instruction level. The path for generating the simulation input is shown in Figure 2. The benchmark programs and a hardware configuration file are required inputs to the simulation path. GCC compiles the benchmark source code into a machine independent intermediate code. The code generation step interprets the intermediate code into the instruction set defined for the superscalar architectures, and outputs sequential code which is independent of the target machine's hardware configuration. The sequential code and the hardware configuration are inputs to the scalable simulator, and the number of cycles taken to execute the program is the output.

4.3 Benchmarks

Ten benchmark programs have been chosen to compare the performance of the VIPER and superscalar processors. Table 3 gives a listing and description of the benchmarks. These benchmarks are integer programs which implement

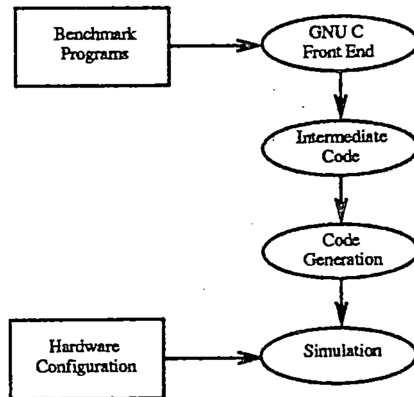

Figure 2: Superscalar Simulation Path

Benchmark	Description
binsearch	Binary search algorithm
bubble	Bubble sorting algorithm
chain	optimal chained matrix multiplication sequence finder
factorial	computes the factorial on numbers from 1 to n
fibonacci	Fibonacci number sequence generator
floyd	Locates shortest path in a graph using Floyd's algorithm
matrix	Matrix multiplication routine
merge	Merge sort algorithm
quicksort	Basic quicksort algorithm
sp	Locates shortest path with Dijkstra's algorithm

Table 3: Benchmark Programs

a variety of basic algorithms.

The dynamic frequency of each instruction class and the run length distribution for the ten benchmarks is shown in Figure 3 and Figure 4, respectively. ALU operations represent 63% of the total instructions. Load/store instructions are 18%, and the remaining 19% of the instructions are control transfer.

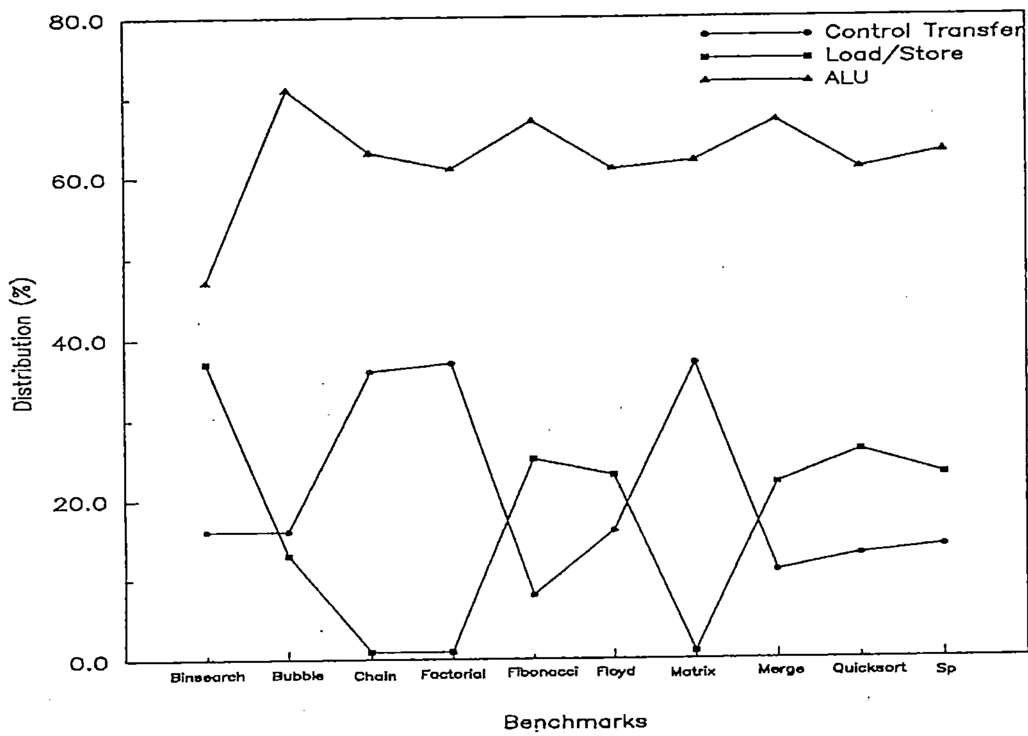

Figure 3: Distribution of Instruction Types in the Benchmark Programs

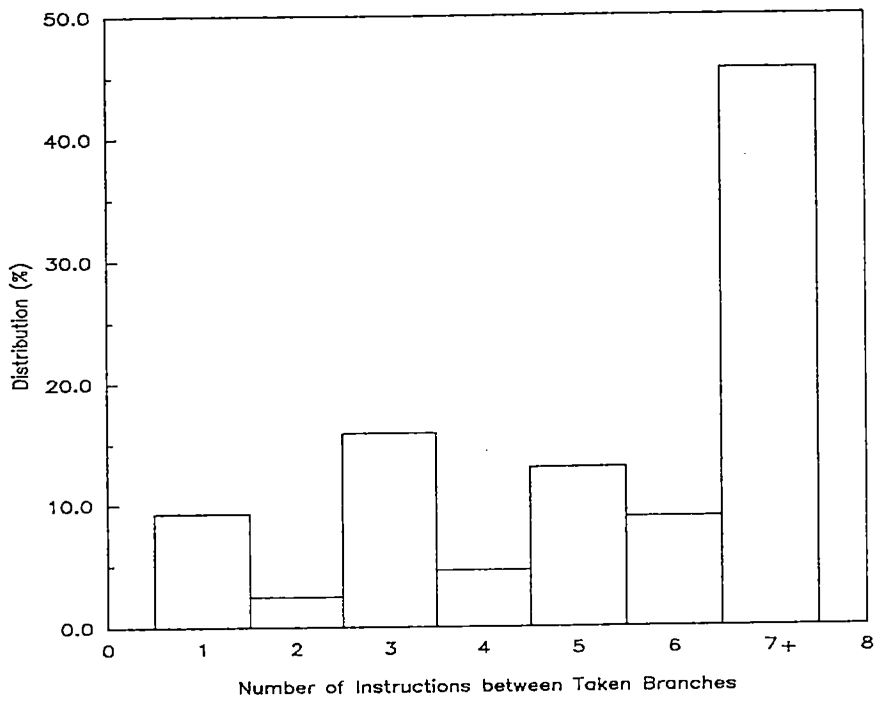

Figure 4: Run Length Distribution of Benchmark Programs

Benchmark	Speedup
binsearch	2.62
bubble	1.54
chain	2.25
factorial	2.37
fibonacci	2.61
floyd	1.94
matrix	2.33
merge	2.13
quicksort	2.04
sp	2.19
Harmonic Mean	2.15

Table 4: Benchmark Performance of VIPER

5 Results

This section presents the results of the simulations for the processors and benchmarks described in the previous sections. For VIPER and the superscalar processors, performance is presented as speedup over the execution time of a scalar processor implementation.

5.1 Comparing Performance

The objective of the simulations is to find a superscalar configuration which achieves comparable performance to VIPER, yet requires the least amount of hardware complexity. The performance of the VIPER processor is given in Table 4 for each of the ten benchmarks.

The parameters which should be minimized to reduce hardware complexity are the following:

- The number of instructions fetched each cycle.
- The number of execution units.
- The size of the instruction window.

Two graphs have been generated to evaluate the above parameters. The graphs show the speedup achieved by several configurations of the superscalar model.

The speedup presented in the graphs is the harmonic mean of the speedups for each benchmark. For these simulations, the instruction cache has a 100% hit ratio, the instruction issue is limited to four instructions per cycle, and the functional units contain only one execution unit.

The first graph, Figure 5 shows the speedup of superscalar processors which can fetch two instructions each cycle. Configurations with different numbers of execution units are distributed along the horizontal axis. Speedup ranges from a low 1.22 to a high of 1.78. Comparing Table 4 and Figure 5 shows the VIPER processor has at least a 17% performance margin over any of the superscalar configurations.

Thus, fetching two instructions per cycle does not supply the superscalar processors with an adequate instruction fetch bandwidth. So, this parameter is changed to a limit of four instructions per cycle to improve performance. Figure 6 graphs the speedup achieved by the superscalar configurations which fetch four instructions each cycle. Points along the horizontal axis represent configurations with different numbers of execution units. Peak performance is achieved by the out-of-order issue scheduling model with a sixteen-entry instruction window; however, the performance of the same scheduling technique with an eight-entry instruction window is nearly the same. The slight performance improvement gained by doubling the instruction window size from 8 to 16 is not justifiable, so the graph shows that the performance close to the VIPER is achieved by a superscalar model with an out-of-order scheduler with an eight entry instruction window. This is an interesting point because a VLIW is designed to exploit a large amount of parallelism by performing global code optimizations, but the superscalar model can achieve nearly the same performance with a very small amount of lookahead ability.

The results have shown that the superscalar model must fetch four instructions each cycle, perform out-of-order instruction issue, and use an eight-entry instruction window to achieve near equal performance to the VIPER. Next, the best execution unit configuration can be determined. From Figure 6, four points along the horizontal axis are shown to have approximately the same peak performance value for the O-W8 curve. Table 5 summarizes these configurations and the speedup achieved by each. This table shows the same performance can be achieved with three or four ALUs. Also, the performance gained by an additional control transfer unit is less than 1% in either case. Considering these tradeoffs, the superscalar model should be configured with three ALUs, two load/store units, and one control transfer unit.

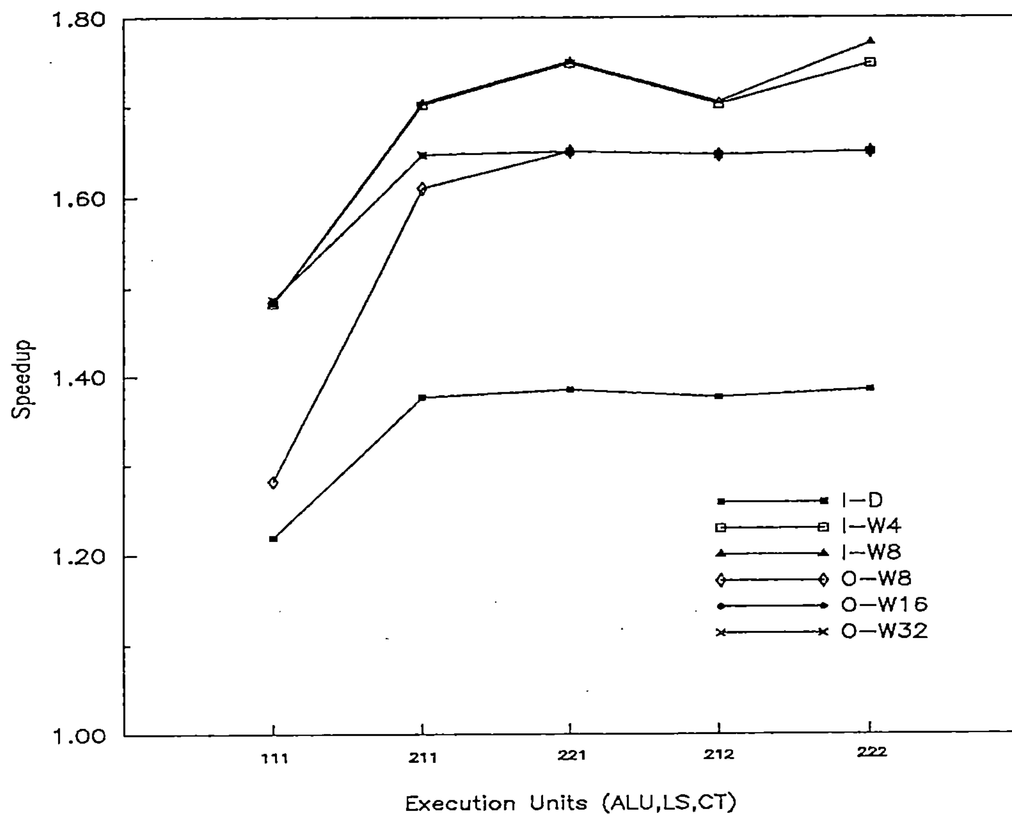

Figure 5: Speedup of Superscalar Processors Fetching 2 Instructions/Cycle

ALUs	LS units	CT units	Speedup
3	2	1	2.10
3	2	2	2.11
4	2	1	2.10
4	2	2	2.11

Table 5: Performance Summary for the O-W8 Scheduling Model

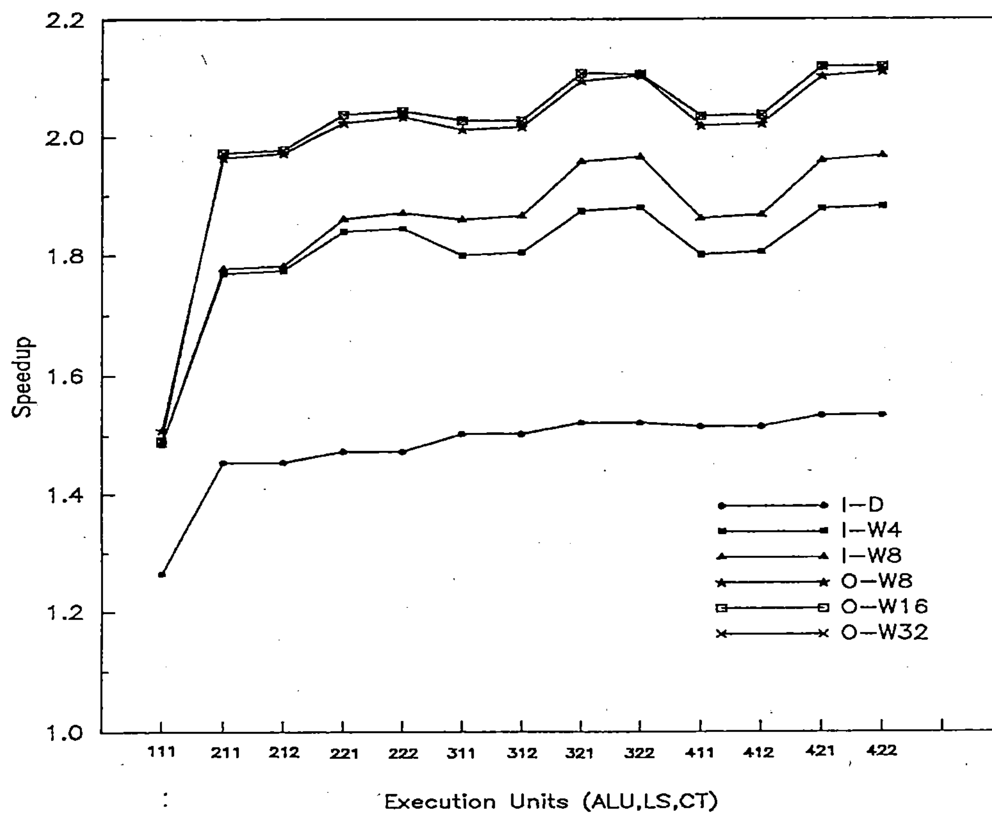

Figure 6: Speedup of Superscalar Processors Fetching 4 Instructions/Cycle

Instruction Window Wize	8
Instructions Fetched per Cycle	4
Instructions Issued per Cycle	3
Instruction Issue Policy	Out-of-order
Arithmetic/Logic Units	3
Load/Store Units	2
Control Transfer Units	1

Table 6: A Superscalar Configuration with Performance Comparable to VIPER

For the previous results, each execution unit was considered to be one functional unit which requires two operand busses from the instruction window to each execution unit. The number of operand busses can be reduced by combining multiple execution units into functional units. The number of functional units must be at least as great as the number of instructions which can be issued each cycle. Currently, the number of instructions issued each cycle is four, so the number of functional units must be at least four. A significant reduction in hardware can be achieved by grouping the six execution units into three functional units, and limiting instruction issue to three instructions. This would reduce the number of ports to the instruction window, the number of busses for distributing instructions and operands to functional units, and the number of bypassing networks to the functional units. The functional units can each be configured with one ALU, and one LS or CT execution unit. Performing the benchmark simulations with this configuration results in a mean speedup of 2.07. This is a performance loss of only 1% which makes this configuration a good design alternative. The resulting superscalar configuration is listed in Table 6. The VIPER has a 4% performance advantage over this superscalar model.

5.2 The Instruction Cache Penalty

Superscalar and VLIW processors respond very differently to an instruction cache miss. Due to dynamic scheduling, when instruction fetching stops as a result of a cache miss, the superscalar processor may take several cycles to issue all of the instructions in the instruction window. If the cache miss penalty can be paid before the window is emptied, then the performance of

the superscalar processor will not be affected by the cache miss. In the case of VIPER, it executes one long instruction each cycle, and must fetch one instruction each cycle to sustain execution. A cache miss causes VIPER to stall until instruction fetch can continue. Therefore, the VIPER processor must pay every cache miss penalty, but the superscalar processor will pay only a fraction of the cache miss penalty.

The benchmark performance of VIPER with a real instruction cache is computed by adding the number of instruction cache stall cycles to the total number of cycles (N_p) required to execute the benchmark without an instruction cache. This sum is used to calculate the new speedup value (*speedup'*) of the processor for each benchmark. The number of instruction cache stall cycles is calculated as follows:

$$Instruction\ cache\ stall\ cycles = A \cdot R \cdot P$$

where

A = Instruction cache accesses per program

R = Instruction cache miss ratio

P = Instruction cache miss penalty

The VIPER processor accesses the cache every cycle so A is equal to N_p, and the total number of cycles required to execute the benchmark including instruction cache stall cycles can be expressed as:

$$N_p' = N_p + N_p \cdot R \cdot P = N_p \cdot (1 + R \cdot P)$$

The new speedup value is computed as:

$$speedup' = \frac{N}{N_p'} = \frac{N}{N_p \cdot (1 + R \cdot P)}$$

where N is the number of cycles required to execute the benchmark on a scalar processor. This equation can be written in terms of the speedup computed without the instruction cache by substituting *speedup* for $\frac{N}{N_p}$. The resulting equation is:

Miss Ratio	3%	4%	5%	6%
Speedup with 2 cycle miss penalty	2.03	1.99	1.95	1.92
Speedup with 3 cycle miss penalty	1.97	1.92	1.87	1.82
Speedup with 4 cycle miss penalty	1.92	1.85	1.79	1.73

Table 7: Speedup of VIPER with Instruction Cache Penalty

$$speedup' = \frac{speedup}{(1 + R \cdot P)}$$

which shows the performance of the VLIW processor degrades linearly as a function of R and P. Table 7 lists the resulting speedup for some typical cache penalties and miss ratios.

The performance of the superscalar processor with an instruction cache must be found through simulation because it cannot be computed from the above equations. The two factors, A and P in the equation given above for calculating instruction cache stall cycles cannot be determined statically for the superscalar processor. The number of instruction cache accesses per program is not equal to the number of cycles required to execute the program as it was for the VLIW processor. A superscalar processor might stall instruction fetch as a result of a non-empty decoder, and these stall cycles cause the number of instruction cache accesses to be less than the number of execution cycles. Also, the average instruction cache miss penalty should be less than the maximum instruction cache miss penalty for the superscalar processor. If the superscalar processor is kept busy issuing instructions from the instruction window during the instruction cache miss cycles, then the miss penalty is zero. The performance of the superscalar processor will only be adversely affected by the instruction cache miss when the instructions in the window are depleted before the end of the miss penalty cycles.

The results of the cached simulations are shown in Figures 7, 8, and 9 along with the values computed for VIPER from Table 7. The values used for the miss ratios range from 3–6% and are shown along the horizontal axis. Figures 7 – 9 show the results for a cache with a two, three, and four cycle miss penalties respectively. These graphs show that the superscalar processor model described in Table 6 outperforms VIPER when the product of the miss ratio and miss penalty exceeds a .12 value. However, if the instruction cache

can be designed with a small miss ratio and miss penalty, then the VIPER processor will continue to perform slightly better than the superscalar model.

6 Conclusion

The simulation results show that VIPER and a superscalar model which performs out-of-order issue can achieve similar performance to the selected benchmarks. It has also been shown that the superscalar processor requires only a small lookahead window to exploit the same amount of fine grain parallelism as the VIPER processor. After taking into effect the instruction cache interface, the superscalar demonstrated an ability to continue instruction issue and execute despite instruction cache misses; whereas, the performance of VIPER degraded linearly with an increasing miss ratio and miss penalty product. As the miss ratio and miss penalty increase, the performance of the superscalar model exceeds the performance of VIPER.

References

[1] A. Abnous. Architectural design and analysis of a VLIW processor. Master's thesis, University of California, Irvine, 1991.

[2] A. Abnous, R. Potasman, N. Bagherzadeh, and A. Nicolau. A percolation based VLIW architecture. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 144–148, 1991.

[3] Robert Colwell, Robert Nix, John O'Donnell, Cavid Papworth, and Paul Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on computers*, 37:967–979, 1988.

[4] Pradeep Dubey and Michael Flynn. Branch strategies: Modeling and optimization. *IEEE Transactions on Computers*, 40(10):1159–1167, October 1991.

[5] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30:478–490, July 1991.

[6] John Hennessey and David Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, 1990.

[7] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, 1991.

[8] Johnny Lee and Alan Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer Magazine*, 17(1):6–22, January 1984.

[9] A. Nicolau. Percolation scheduling: A parallel compilation technique. Technical Report 85-678, Cornell Universisty, 1985.

[10] R. Potasman. *Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs.* PhD thesis, Universisty of California, Irvine, 1991.

[11] James Smith and Andrew Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[12] Gurndar Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[13] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, January 1967.

[14] Shlomo Weiss and James Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, c-33(11):1013–1022, November 1984.

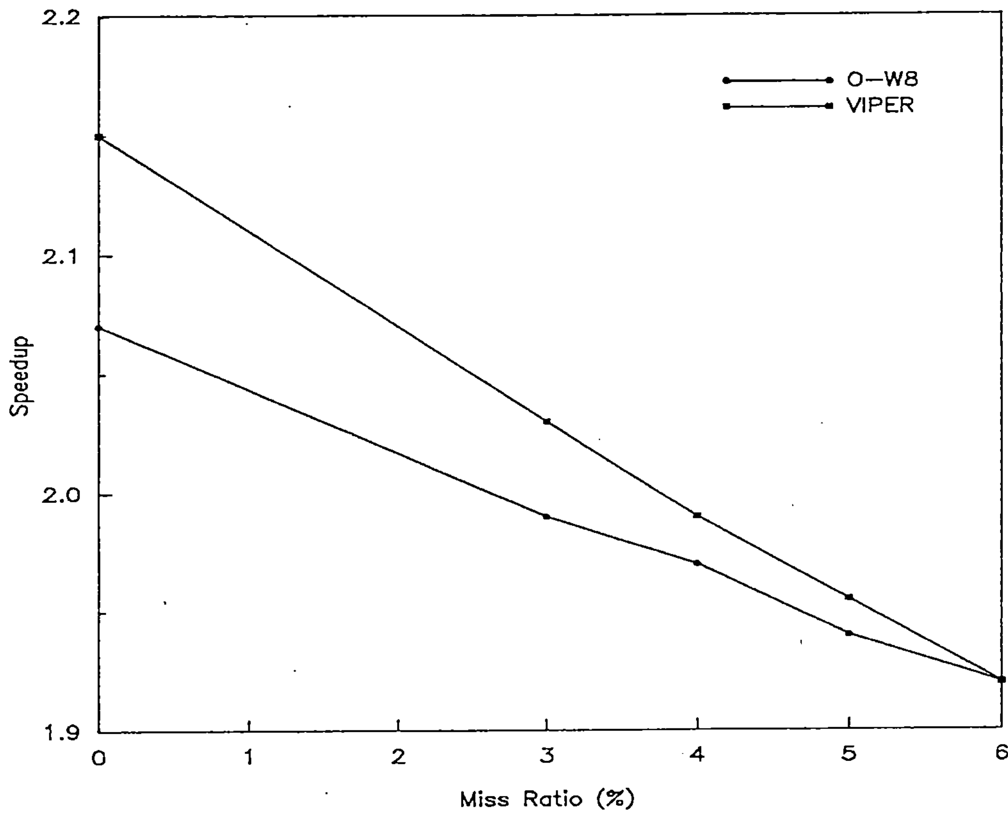

Figure 7: Speedup with Instruction Cache Miss Penalty = 2

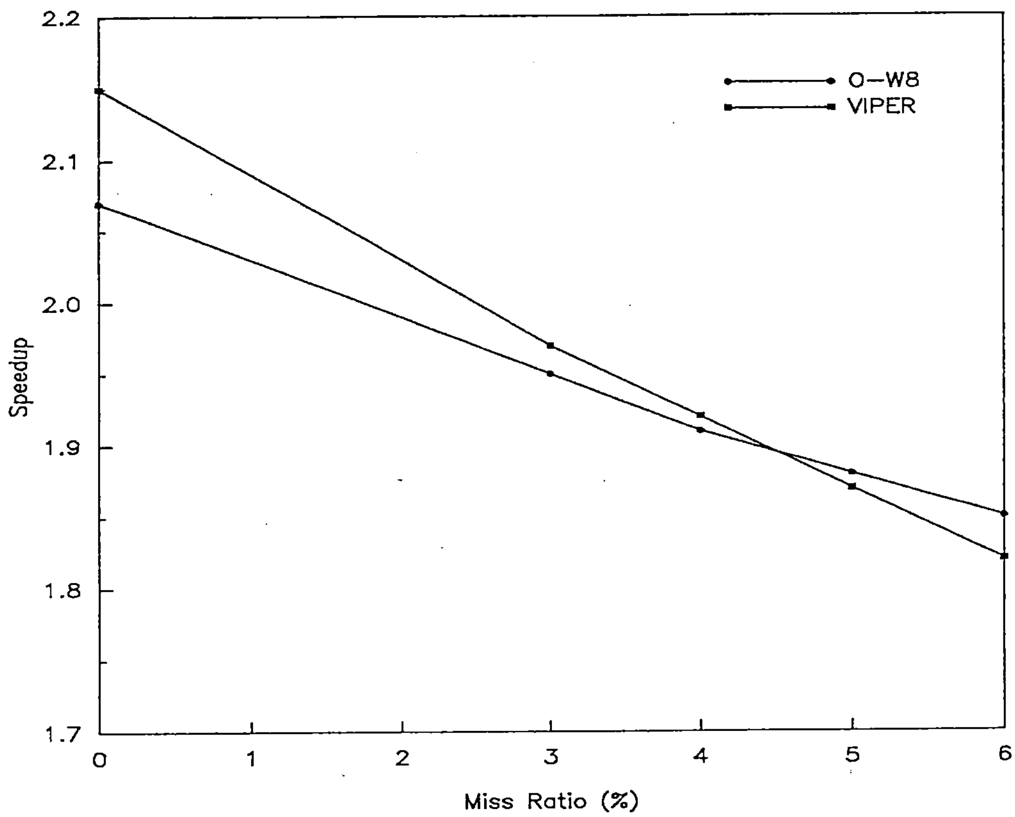

Figure 8: Speedup with Instruction Cache Miss Penalty = 3

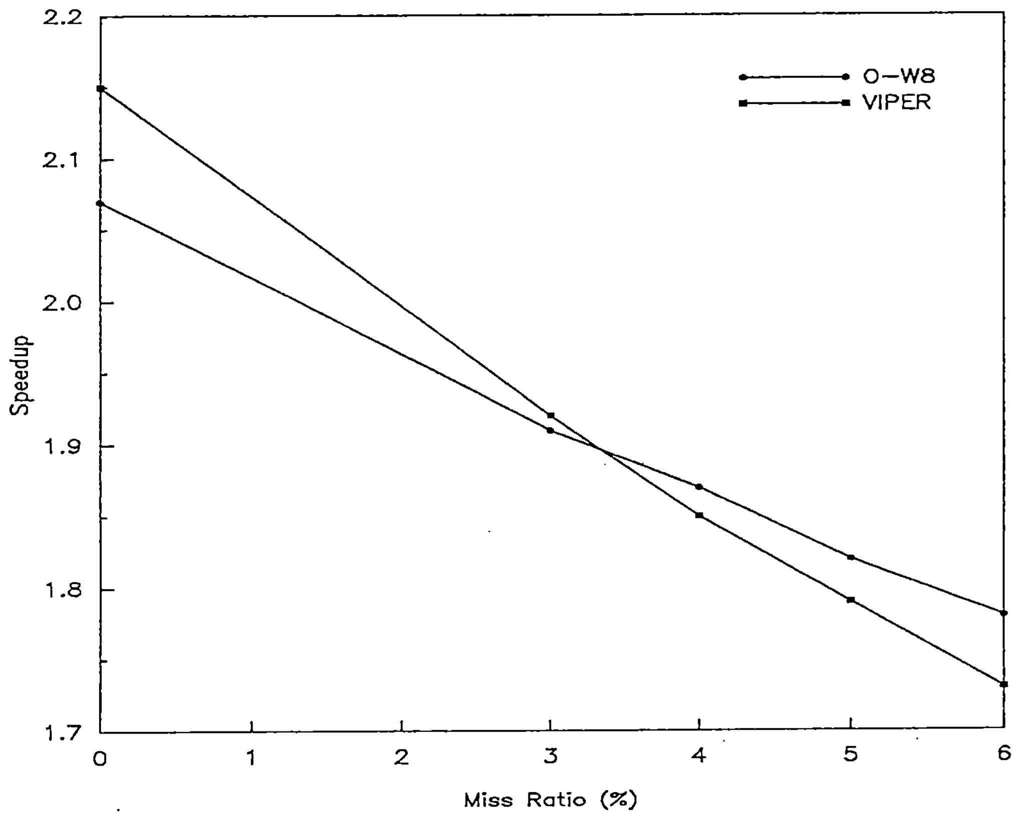

Figure 9: Speedup with Instruction Cache Miss Penalty = 4